Joonas Kajava

Luotettavan tietoliikenneprotokollan suunnittelu ja toteutus Rustilla

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto ja viestintätekniikka

Insinöörityö

28.5.2024

Tiivistelmä

Tekijä:	Joonas Kajava
Otsikko:	Luotettavan tietoliikenneprotokollan suunnittelu ja toteutus Rustilla
Sivumäärä:	31 sivua
Aika:	28.5.2024
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Pelisovellukset
Ohjaajat:	Lehtori Miikka Mäki-Uuro

Tässä insinöörityössä esitetään luotettavan tietoliikenneprotokollan suunnittelu ja toteutus, minkä nimeksi tuli RRTP-protokolla. Protokollaa varten tehtiin ohjelmakirjasto Rust-kielelle, mistä lopputuloksena syntyi RRTP-protokollan toteuttava ohjelmakirjasto. Se yhdistää liukuvan ikkunan ja muita tekniikoita luotettavan yhteyden muodostamiseen.

Protokollan toimivuus testattiin demosovelluksella, joka kehitettiin tässä työssä käyttämällä Tauri-ohjelmistokehystä. Tauri-ohjelmistokehys hoitaa käyttöliittymän piirtämisen. Sovellus hyödyntää RRTP-ohjelmakirjastoa yhteyden muodostamiseen ja tiedon välittämiseen kahden tietokoneen välillä.

Kehityksen aikana toteutettiin suorituskykyanalyysejä, joissa vertailtiin tietorakenteiden nopeuksia protokollan käyttötarkoituksissa. Analyysin perusteella todettiin, että LinkedList-tietorakenne ei ole sopiva tähän projektiin. Toteutukseen valittiin Vec-tietorakenne, jonka suorituskyky oli erinomainen ja joka sopii parhaiten projektin käyttötarkoituksiin.

Lopuksi pohdittiin työtapoja ja työkaluja, joita käytettiin kehityksessä. Työssä käytettiin Git-versionhallintaa ja kehitysympäristö koostui Windows-tietokoneesta sekä RustRoverista, joka on Rust-kieleen erikoistunut ohjelmointiympäristö.

Avainsanat: Rust, UDP, suorituskyky, liukuva ikkuna, protokolla

---

Abstract

Author:	Joonas Kajava
Title:	Design and implementation of a reliable communication protocol in Rust
Number of Pages:	31 pages
Date:	28 May 2024
Degree:	Bachelor of Engineering
Degree Program:	Information and Communication Technology
Professional Major:	Game Applications
Supervisors:	Miikka Mäki-Uuro, Senior Lecturer

This bachelor's thesis showcases the design and implementation of a reliable communication protocol, which was named RRTP protocol. A library for the Rust programming language was created for the protocol. the end product was a library that implements the RRTP protocol, which combines a sliding window and other techniques for a reliable connection.

The robustness of the protocol was tested using a demo application that was also developed during this project. The application was created using the Tauri framework, which handles drawing the user interface. The demo application utilizes the RRTP library to handle the connection and data transfer between two computers.

Performance analysis was carried out for data structures used in the protocol during the development. Analysis indicated that the LinkedList data structure is not suitable for this project and that the Vec data structure is more appropriate for this project.

The thesis concludes with a reflection section that discusses the development practices and tools used in this project. Git was used as the version control system and the development environment was a Windows machine with RustRover, which is an integrated development environment for the Rust language.

Keywords: Rust, UDP, performance, sliding window, protocol

Sisällys

Lyhenteet

ACK: *Acknowledgement*. Vastaanottajan kuittaus saapuneesta paketista.

API: *Application Programming Interface*. Komponenttien välinen ohjelmointirajapinta.

ARQ: *Automatic Repeat Request*. Automaattinen uudelleenlähetys.

DI: *Dependency Injection*. Riippuvuuksien injektointi, jonka avulla tarvittavat riippuvuudet on saatavilla tietyissä aliohjelmissa.

IP: *Internet Protocol*. IP-pakettien toimituksesta huolehtiva protokolla.

IPC: *Inter-Process Communication*. Prosessien välinen kommunikaatio.

MIME: *Multipurpose Internet Mail Extensions*. MIME-tyyppi kertoo tiedostomuodon standardin mukaisella tavalla.

NACK: *Negative Acknowledgement*. Negatiivinen vastaanottajan kuittaus.

NIC: *Network Interface Controller*. Verkkosovitin, jonka tehtävä on muodostaa yhteys lähiverkkoon.

RPC: *Remote Procedure Call*. Etäproseduurikutsu, jonka avulla asiakasohjelma voi lähettää kutsuviestin palvelinohjelmalle.

TCP: *Transmission Control Protocol*. Tietoliikenneprotokolla, joka muodostaa jatkuvan yhteyden tietokoneiden välille.

UDP: *User Datagram Protocol*. Yksinkertainen tietoliikenneprotokolla, joka ei tarjoa virheenkorjausta.

1 Johdanto

Tässä työssä suunniteltiin ja toteutettiin tietoliikenneprotokolla Rust-kielelle. Tietoliikenneprotokolla käyttää UDP-protokollaa pohjana ja laajentaa sitä uusilla ominaisuuksilla. UDP on yksinkertainen tietoliikenneprotokolla, joka ei tarjoa pakettien järjestämistä tai kattavaa virheenkorjausta. Tarkoitus on luoda pohja yrityksille ja kehittäjille tietoliikenneprotokollan toteutukseen. Työ on luonteeltaan tutkiva ja ohjeistava, joka selventää reaaliaikaisien sovelluksien toteutusta.

Protokollaa varten tehtiin RRTP-ohjelmistokirjasto käyttäen Rust-ohjelmointikieltä, joka soveltuu erinomaisesti järjestelmäohjelmointiin. Rust-kieli mahdollistaa laitteiston matalan tason hallinnan samaan tapaan kuin C-ohjelmointikieli. Iso etu Rust-kielessä on, että se käyttää automaattista muistinhallintaa, joka ei käytä roskankerääjää, vaan muistin vapautus tapahtuu omistus- ja lainausjärjestelmän sääntöjen mukaisesti. Nämä säännöt takaavat Rust-kielen muistiturvallisuuden, joka on todella tärkeää ohjelman toimivuuden kannalta. Rust-kieli tarjoaa myös monia moderneja kieliominaisuuksia kuten tyyppipäättely (engl. type inference), hahmonsovitus (engl. pattern matching) ja piirteet (engl. traits) [1].

Johdannon jälkeen siirrytään käsittelemään RRTP-protokollan toteuttavan kirjaston toimintaa. RRTP-kirjasto on neljään osaan jaettu kokonaisuus, joka käyttää säikeitä ja kanavia sisäiseen tiedon siirtoon. Kirjaston avulla ohjelmat, kuten tässä työssä kehitetty demosovellus pystyvät kommunikoimaan viestien ja tiedostojen avulla. Toteutuksen läpikäynti alkaa paketin vakioiden ja rakenteen määrityksellä, jonka jälkeen siirrytään käsittelemään kirjaston arkkitehtuuria.

Luotettavuus tässä tietoliikenneprotokollassa tulee liukuvasta ikkunasta. Se sallii pakettien lähettämisen ja vastaanottamisen epämääräisessä järjestyksessä sekä takaa viestien perille pääsyn aikakatkaisun avulla. Työssä liukuvan ikkunan toiminta esitetään lähettäjän ja vastaanottajan näkökulmista, jossa toiminta havainnollistetaan diagrammien ja koodin avulla. Tämän jälkeen keskustellaan tarkemmin valikoiva toisto ARQ-protokollasta.

Viestien käsittelyssä esitetään eri tietomuodot, joita voidaan käyttää tiedonsiirrossa. Tietomuotojen määritys on tärkeää tiedonsiirrossa, sillä vastaanottajan ja lähettäjän täytyy olla yhteisymmärryksessä niistä. Demosovelluksessa tiedonsiirto tapahtuu käyttäen binäärimuotoa, mutta skeemapohjaisen tiedon käyttäminen on mahdollista. Skeemapohjaisen tiedon ja binääritiedon erot esitetään esimerkkien avulla.

Viimeisenä RRTP-protokollasta esitetään virheenkorjaus, johon kuuluu 16-bittinen tarkistussumma. Sen sijoitus paketissa havainnollistetaan taulukon ja koodin avulla. Lopuksi tarkistussumman muodostus esitetään esimerkkikoodin avulla.

RRTP-kirjastoa varten toteutettiin myös demosovellus, joka tarjoaa toiminnot viestien ja tiedostojen lähettämistä varten. Demosovelluksesta esitetään sen riippuvuudet eri kirjastoihin sekä Rust-prosessin ja käyttöliittymän välisen kommunikaation.

Tämän työn tavoitteena oli tutkia tietoliikenneprotokollien toimintaa ja niiden toteutusta. On usein kannattavaa kehittää sovelluksen tarpeisiin mukautettu protokolla, mikäli kaistanleveys ja vasteaika merkitsee huomattavasti.

2 RRTP-protokolla

TCP- ja UDP-protokollat ovat kaksi yleisintä TCP/IP-protokollaa. TCP-protokolla on jatkuvan yhteyden muodostava tietoliikenneprotokolla, jota käytetään usein vakaata ja luotettavaa yhteyttä vaativissa palveluissa, kun taas UDP on usein paras reaaliaikaisia sovelluksia varten. Tämä johtuu siitä, että TCP-paketin mukana liikkuu paljon varattuja tavuja, joita ei käytetä ollenkaan.

UDP:n etu reaaliaikaisessa kommunikaatiossa on, että se on kevyt ja nopea. Se ei kuitenkaan tarjoa kattavaa virheenkorjausta, järjestystä tai ruuhkanhallintamekanismeja. Tarkistussumma on ainut virheentarkistus, minkä UDP tarjoaa. Se ei kuitenkaan yksin riitä vakaaseen yhteyteen, vaan lisäksi täytyy huomioida pakettien järjestys. Tämän takia liukuva ikkuna ja automaattinen uudelleenlähetys(AQR) ovat tärkeitä tekniikoita kommunikaation luotettavuudessa, sillä niiden avulla paketit voivat liikkua verkon yli epämääräisessä järjestyksessä. Pakettien uudelleen järjestäminen vastaanottaessa vaatii järjestysnumeron. UDP sisältää lähettäjän ja vastaanottajan portit, datan koon, tarkistussumman ja datan. [2.]

Tässä luvussa esitetään RRTP-protokollan toteutus, joka on reaaliaikaiseen kommunikointiin erikoistunut tietoliikenneprotokolla.

Protokollan paketti sisältää ohjausbittejä, jotka ilmaisevat tärkeää tietoa paketista. Näiden bittien avulla vastaanottaja tietää, milloin viimeinen paketti on saapunut perille ja lähettäjä saa tiedon kuittauksesta. Liukuvaa ikkunaa hyödynnetään paketin järjestämiseen ja samanaikaiseen lähettämiseen. Vastaanottaja ja lähettäjä voivat välittää paketteja keskenään järjestyksestä riippumatta, sillä ikkunan

puskuri järjestää paketit niiden järjestysnumeron mukaan.

2.1 Paketti

Paketti, joka on havainnollistettu taulukossa 2, perustuu kevyesti TCP:hen ja on rakennettu käyttäen UDP:ta pohjana. Paketin rakenne on minimaalinen ja yksinkertainen, mikä sisältää tarvittavat kentät luotettavaan yhteyteen. Nämä kentät ovat järjestysnumero, ohjausbitit ja kentät datan hallintaa varten. Pakettiin sisältyy yksi tavun kokoinen varattu kenttä, jota voidaan käyttää tulevaisuudessa. 6 ohjausbittiä ei ole käytössä, joita voidaan käyttää myöhemmin uusiin ominaisuuksiin.

Vakiot

Taulukkoon 1 on koostettu ne vakiot, jotka ovat tärkeitä protokollan toiminnan kannalta. Vastaanottajalla ja lähettäjällä täytyy olla samat vakiot käytössä, jotta pakettien muodostaminen ja lukeminen onnistuu. Eroavaisuudet näissä vakioissa vastaanottajan ja lähettäjän välillä voivat tarkoittaa, että vastaanottaja aloittaa datan lukemisen väärästä kohdasta.

Taulukko 1: Vakiot ja niiden arvot.

Vakion nimi	Lyhenne	Arvo	Yksikkö
MAX_DATA_SIZE	D_s	128	Tavu
SEQ_NUM_SIZE	S_s	4	Tavu
CONTROL_BITS_SIZE	C_s	1	Tavu
RESERVED_SIZE	R_s	1	Tavu
DATA_LENGTH_SIZE	DL_s	1	Tavu
DATA_OFFSET_SIZE	DO_s	1	Tavu
OPTION_KIND_SIZE	OK_s	1	Tavu
OPTION_LENGTH_SIZE	OL_s	1	Tavu
OPTION_DATA_SIZE	OD_s	4	Tavu
MAX_OPTION_COUNT	MO_c	4	Määrä

Näiden vakioiden perusteella voidaan laskea tärkeitä muuttujia, jotka on määritelty kaavojen 1-4 mukaisesti.

$$\text{MIN\_FRAME\_SIZE} = F_{min} = S_s + C_s + R_s + DL_s + DO_s \quad (1)$$

$$\text{MAX\_FRAME\_SIZE} = F_{max} = F_{min} + MO_c(OK_s + OL_s + OD_s) + D_s \quad (2)$$

$$\text{Frame size} = F_s \quad (3)$$

$$\text{Option size} = O_s = F_s - F_{min} - D_s \quad (4)$$

Vakikoiden ja laskettujen muuttujien perusteella voidaan rakentaa paketin kehys, joka on havainnollistettu taulukossa 2. Kehys määrittelee jokaisen arvon sijainnin ja pituuden paketissa. Näin lähettäjällä ja vastaanottajalla on yhtenäinen ymmärrys paketin rakenteesta.

Taulukko 2: Paketin rakenne

Offsets		0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
0	0	Sequence number																															
4	32	RES	RES	RES	RES	RES	RES	EOM	ACK	Reserved								Data Length								Data Offset							
8	64	Options																															
...	...																																
Data Offset		Data																															
...	...																																

Lähdekoodissa 1 määritetään tietorakenne *Frame*, joka sisältää kaiken paketissa olevan tiedon. [u8; MAX_FRAME_SIZE] on taulukkotyyppi, joka koostuu tavuista ja on kooltaan yhtä suuri kuin F_{max}, joka on määritelty kaavassa 2. data_length sisältää tiedon pakettiin tallennetun datan koosta, kun taas options_size sisältää tiedon paketin asetuksien koosta, mitä käytetään datan asettamisen oikeaan paikkaan ja *'Data Offset'* kentän asettamiseen.

Paketti on määritelty seuraavasti:

```
1 pub struct Frame {
2     frame: [u8; MAX_FRAME_SIZE],
3     data_length: usize,
4     options_size: usize,
5 }
```

Lähdekoodi 1: Paketin rakenne

usize vastaa kohdearkkitehtuurin muistiosoitteen kokoa. 32-bittisessä tietokoneessa usize vastaa 4 tavua ja 64-bittisessä kohteessa 8 tavua [3, *usize*].

Tietojenkäsittely paketissa

Paketti on ohjelman muistissa yhtenä kokonaisena taulukkona, joka koostuu tavuista. Taulukko sisältää tietoja 16-bittisessä ja 32-bittisessä muodossa. Nämä tiedot jaetaan tavuihin lähettämistä varten.

Lähdekoodissa 2 järjestysnumero muutetaan laskevaan tavujärjestykseen käyttäen Rust-standardikirjaston to_be_bytes()-aliohjelmaa [3, *u32*]. Tämän jälkeen tulos kopioidaan paketin ensimmäisen 4 tavun tilalle.

```
pub fn set_sequence_number(&mut self, sequence_number: u32) {
    let net_sequence_number = sequence_number.to_be_bytes();
    self.frame[SEQUENCE_NUMBER_OCTET..4]
    .copy_from_slice(&net_sequence_number);
}
```

Lähdekoodi 2: Järjestysnumeron asettaminen pakettiin

Oikea tavujärjestys on erittäin tärkeä osa verkon ylitse tehtävää tiedonsiirtoa. Protokollaa käyttävien tietokoneiden täytyy olla yhteisymmärryksessä siitä, mitä tavujärjestystä tiedonsiirrossa käytetään. Laskeva tavujärjestys on yleisin järjestys, joten sitä käytettiin myös tässä protokollassa.

Mikäli tavujärjestystä ei oteta huomioon tiedonsiirrossa ja vastaanottajan sekä lähettäjän tietokoneet käyttävät eri tavujärjestystä, lukujen muuntaminen tavutaulukosta primitiiviseksi luvuksi johtaa vääriin tuloksiin [4].

Ohjausbitit

Ohjausbitit ilmaantuvat paketissa yhtenä tavuna, joista jokainen bitti vastaa tiettyä totuusarvoa.

Taulukon 3 määrittämät ohjausbitit antavat vastaanottajalle tärkeää tietoa paketista. Kuittausbitti ilmoittaa lähettäjälle, että vastaanottaja on saanut paketin onnistuneesti. Päätebitti ilmoittaa vastaanottajalle, että kyseinen paketti on pakettiryhmän viimeinen ja ryhmä on valmis koottavaksi, kun kaikki sitä edeltävät paketit ovat saapuneet.

Vastaanottaja tarkistaa ohjausbittien olemassaolon bittioperaatiolla lähdekoodin 3 mukaisesti.

Taulukko 3: Ohjausbitit ja niiden arvot

Nimi	Lyhenne	Binääriarvo
Kuittaus	ACK	00000001
Pääte	EOM	00000010

```
if control_bits & 00000001 == 00000001 {
    // Kuittausbitti löytyy
}
```

Lähdekoodi 3: Kuittausbitin tarkistus ohjausbiteistä

2.2 Arkkitehtuuri

RRTP-protokollan toteuttava kirjasto on jaettu neljään osaan. Nämä osat kommunikoivat keskenään kanavien avulla. Kirjaston arkkitehtuuri on havainnollistettu kuvassa 1.

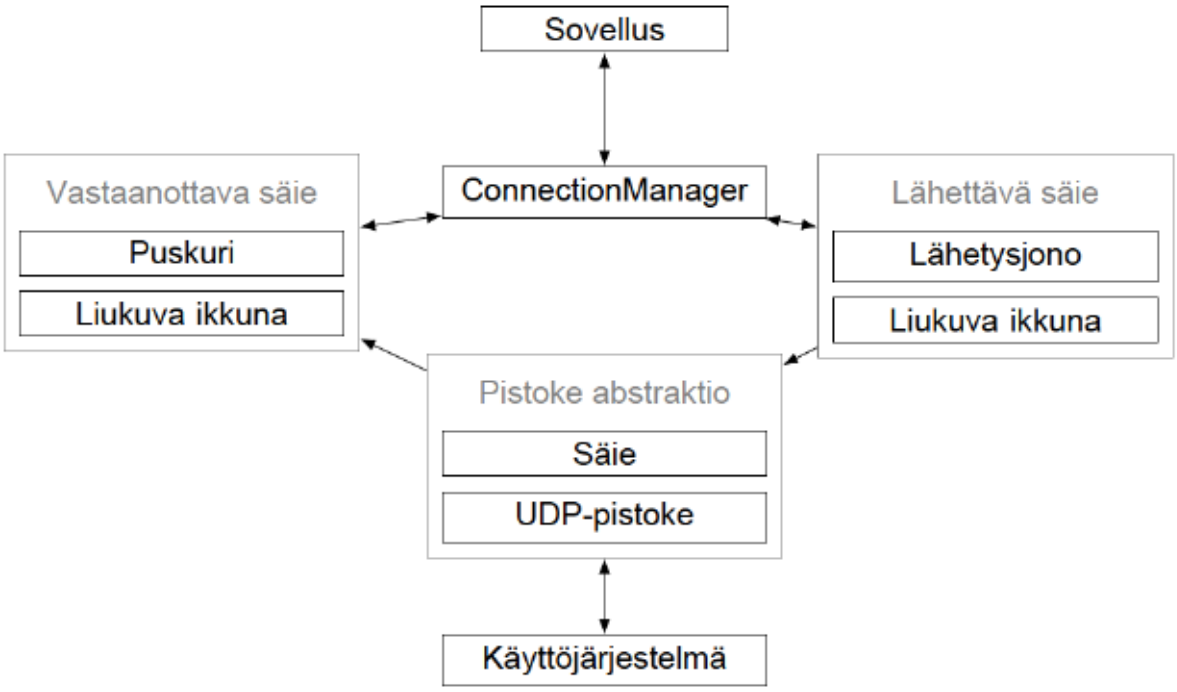
Kirjaston ydin on ConnectionManager, joka on vastuussa yhteyksien hallinnasta. Sen tehtävä on hoitaa vastaanottavaa säiettä ja lähettävää säiettä. Kun sovellus haluaa käynnistää yhteyden, ConnectionManager luo pistokkeen sovelluksen antamaan osoitteeseen. Lähetys- ja vastaanottosäikeet käynnistyvät, kun pistokkeen luonti onnistuu. ConnectionManagerin rakenne on määritelty lähdekoodin 4 mukaisesti.

```
pub struct ConnectionManager {
    listener_thread: Option<thread::JoinHandle<()>>,
    transmitter_thread: Option<thread::JoinHandle<()>>,
    socket: Arc<SocketAbstraction>,
    message_sender: SyncSender<Vec<u8>>,
}
```

Lähdekoodi 4: ConnectionManagerin rakenne

ConnectionManagerilla on omistajuus vastaanottaja- ja lähettäjäsäikeisiin. Option<...> määrittelee valinnaisen arvon, joka tarkoittaa, että kyseinen muuttuja ei välttämättä sisällä käyttökelpoista arvoa. Tämä vastaa useissa ohjelmointikielissä null-arvoa [1, luku 6.1]. Näitä kahta muuttujaa käytetään säikeiden sulavaan sulkemiseen, kun ConnectionManager pudotetaan muistista. ConnectionManager toteuttaa Drop-ominaisuuden, joka liittää vastaanottaja- ja lähettäjäsäikeet.

Näin ohjelma jää odottamaan, että nämä kaksi säiettä sulkeutuvat oikein [3, *Join-Handle*].



Kuva 1: Arkkitehtuuridiagrammi

```
impl Drop for ConnectionManager {
    fn drop(&mut self) {
        if let Some(x) = self.listener_thread.take() {
            x.join().unwrap();
        }
        if let Some(x) = self.transmitter_thread.take() {
            x.join().unwrap();
        }
    }
}
```

Lähdekoodi 5: Drop-ominaisuuden toteutus ConnectionManagerille

2.3 Liukuva ikkuna

Liukuva ikkuna on tekniikka, joka sallii pakettien lähettämisen ja vastaanottamisen epämääräisessä järjestyksessä. Kirjasto käyttää kahta liukuvaa ikkunaa, jotka on

havainnollistettu arkkitehtuuri kuvassa 1. Lähettäjän ja vastaanottajan ikkunat jakavat suurimman osan toiminnallisuuksista Window-toteutuksen kautta, joka on esitetty lähdekoodissa 6.

```
pub struct Window {
    frame_status: Vec<bool>,
    window_size: u32,
    window_left_edge: u32,
}
```

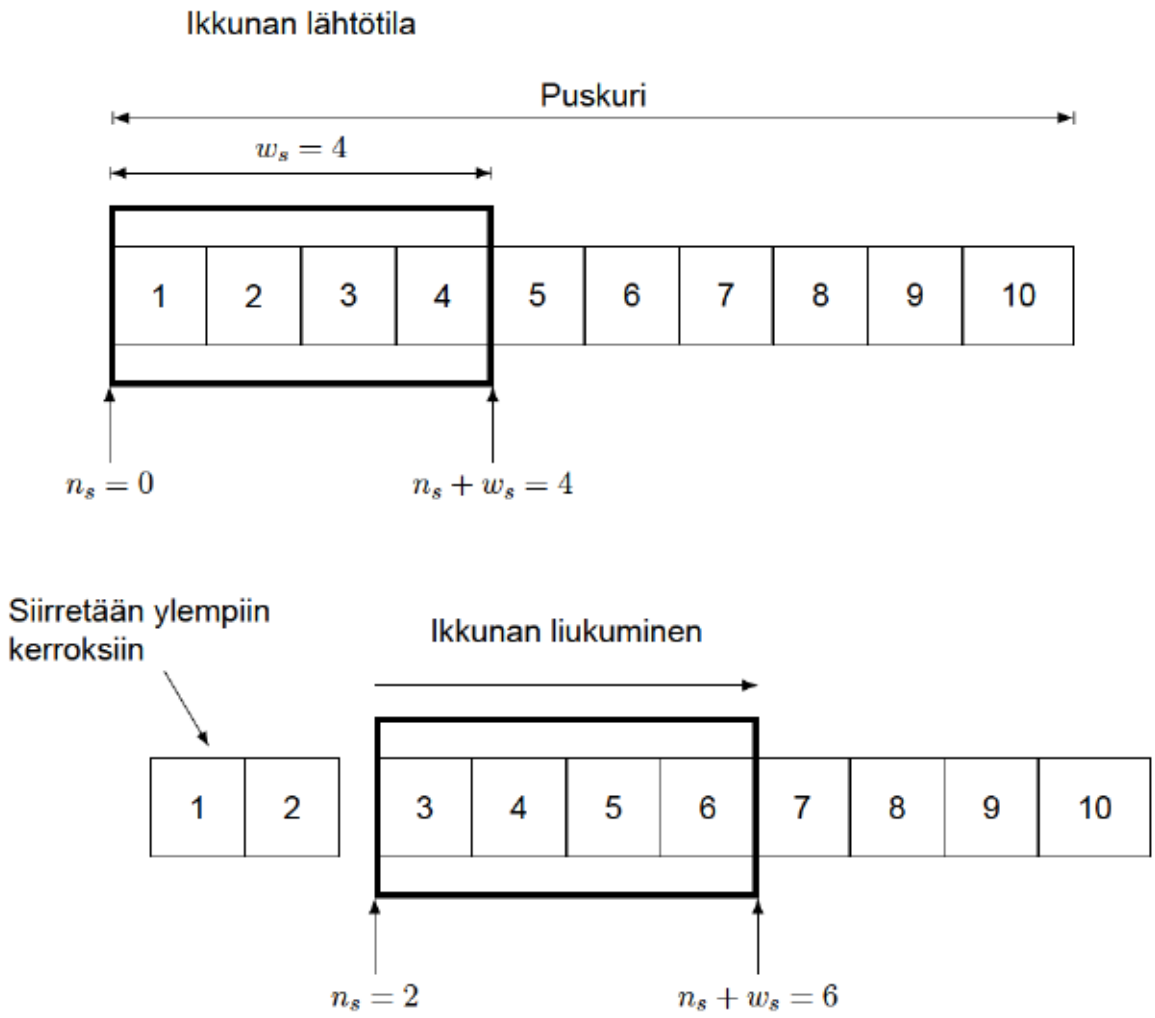
Lähdekoodi 6: Ikkunan rakenne

frame_status-kentällä on kaksi tarkoitusta riippuen siitä, onko kyseessä vastaanotto- tai lähetystila. Kummassakin tilanteessa kenttä on aina ikkunan w_s kokoinen. Vastaanottotilanteessa kenttä pitää kirjaa onnistuneiden pakettien vastaanottamisesta, kun taas lähetystilanteessa kenttä pitää kirjaa paketeista, jotka ovat saapuneet onnistuneesti toiseen tietokoneeseen.

window_size-kenttä on aina w_s kokoinen. Ikkunan kokoa on mahdollista muuttaa käyttämällä set_window_size-aliohjelmaa, joka muuttaa tämän kentän arvoa ja muuttaa frame_status-taulukon kokoa.

window_left_edge vastaa muuttujaa n_s. Vastaanottajan tilanteessa se sisältää pienimmän järjestysnumeron, jota ei ole vastaanotettu, kun taas lähettäjän tilanteessa se sisältää pienimmän järjestysnumeron, joka ei ole vielä saapunut vastaanottajalle. Muuttujan n_s kasvaminen tarkoittaa ikkunan liukumista eteenpäin, joka on havainnollistettu kuvassa 2.

Tässä ohjelmassa puskurit ovat toteutettu käyttämällä kasa-allokoituja vektoreita. Vec on yleisesti paras tietorakenne tämän tyylisiä operaatioita varten. Vertailu muihin tietorakenteisiin löytyy osiosta 4.1.



Kuva 2: Liukuvan ikkunan toiminta

Ikkunan siirto vastaanottaessa

Vastaanottajan ikkuna määritellään lähdekoodin 7 mukaisesti. Se sisältää yleisen Window-toteutuksen ja buffer-kentän, jonka tarkoitus on toimia puskurina, kunnes ikkunan liukuminen aiheuttaa valmiiden pakettien siirtymisen ylempiin kerroksiin.

```
pub struct ReceiverWindow {
    inner_window: Window,
    buffer: Vec<Option<Frame>>,
}
```

Lähdekoodi 7: Vastaanottajan ikkunan rakenne

Rust-kielessä ei pysty tekemään perintää samantyylisesti kuin yleisissä olio-ohjelmointikielissä. Tämän takia käytetään koostumus-suunnittelutapaa (engl.

Composition) [5].

Vastaanottava ikkuna pitää muistissa pienimmän järjestysnumeron n_s, minkä se on vastaanottanut. Vastaanottava ikkuna hylkää kaikki paketit, joiden järjestysnumero on ikkunan ulkopuolella $n_x < n_s$ tai $n_x > n_s + w_s$. Jos järjestysnumero on ikkunan sisällä, se hyväksytään ja merkitään vastaanotetuksi. Mikäli $n_x = n_s + 1$, ikkunaa voidaan siirtää eteenpäin. Siirto tapahtuu kulkemalla hyväksyttyjen järjestysnumeroiden listaa eteenpäin, kunnes vastaan tulee järjestysnumero, jota ei ole vielä vastaanotettu. Kulkemisen jälkeen tapahtuneiden askelien määrä lisätään muuntujaan n_s.

Tässä toteutuksessa n_s löytyy yleisen ikkunan (lähdekoodi 6) sisältä kentästä window_left_edge.

Vastaanottajan toiminta

Datan vastaanottaminen tapahtuu vastaanottajasäikeessä, joka käynnistetään yhteyden muodostamisessa. Säie pyörii ikuisessa silmukassa, jossa ensimmäinen operaatio on ikkunan siirto eteenpäin.

Lähdekoodi 8 esittää ikkunan siirtämistä eteenpäin, jonka tehtävä on kasvattaa n_s-arvoa ja ilmoittaa ylemmälle kerrokselle siirron määrä. frame_status viittaa onnistuneesti lähetettyihin tai vastaanotettuihin paketteihin.

```
pub fn shift_window(&mut self) -> usize {
    let mut shift_amount = 0usize;
    for e in self.frame_status.iter() {
        if *e {
            shift_amount += 1;
        } else {
            break;
        }
    }
    self.frame_status.drain(0..shift_amount);
    self.window_left_edge += shift_amount as u32;
    shift_amount
}
```

Lähdekoodi 8: Ikkunan siirto

Lähdekoodissa 9 data puretaan puskurista ja puskuria siirretään eteenpäin. Tämä operaatio tuottaa listan paketeista, jotka ovat saapuneet onnistuneesti ja ovat oikeassa järjestyksessä. Kuva 2 havainnollistaa puskurin siirron ja siitä syntyvät paketit. Lista käsitellään flatten-aliohjelmalla, joka tuottaa listan, missä Option on suodatettu pois. Lopuksi puskurin kapasiteettia kutistetaan. Käsitelty lista talletaan ja siirretään käsiteltäväksi ylempiin kerroksiin.

```
pub fn shift_window(&mut self) -> Vec<Frame> {
    let shift_amount = self.inner_window.shift_window();
    let shifted_frames = self.buffer.drain(0..shift_amount);
    let result = shifted_frames.into_iter().flatten().collect();
    self.buffer.shrink_to_fit();
    result
}
```

Lähdekoodi 9: Datan purkaminen puskurista

Jokainen valmis paketti käsitellään ja niiden sisältö lisätään ylemmän kerroksen puskuriin, minkä jälkeen ohjausbitit luetaan paketista kappaleen 2.1 mukaisesti. Vastaanotetusta paketista välitetään tieto ylempiin kerroksiin (kuten sovellukseen), jotka voivat käyttää tätä tietoa esim. latausilmaisimen tekemiseen.

Mikäli paketti on viimeinen viesti, puskuri siirretään uuteen listaan ja välitetään ylemmille kerroksille.

Lähettäjän toiminta

Dataa voidaan lähettää käyttämällä ConnectionManagerista löytyvällä kanavalla. Lähettäjäsäie seuraa tätä kanavaa ja käsittelee sieltä tulevat viestit.

Aluksi viestit leikataan osiin, jossa jokainen osa on kooltaan D_s tai pienempi. Tämän jälkeen valmis kehys lisätään jonoon, josta se lähetetään käyttäen lähetys säikeen liukuvaa ikkunaa.

Lähetyssäie lukee data_queue-taulukkoa ja aloittaa datan lähettämisen, kun se sisältää kehyksiä. Taulukko käsitellään ja jokaisen kehyksen tilanne luetaan. Jokaisella kehyksellä on kolme mahdollista tilaa: ei lähetetty, lähetetty ja kuitattu. Pakettien lähtötilanne on ei lähetetty. Mikäli taulukon läpikäydessä kohdataan jo aikaisemmin lähetetty kehys, sen lähetysajankohtaa verrataan aikakatkaisu muuttujaan. Aikakatkaisun avulla lähettäjä voi lähettää kehyksen uudelleen, mikäli lä-

hetys ajankohdasta on kulunut tarpeeksi aikaa. Lähetyksen jälkeen kehyksen tila päivitetään lähetetty-tilaan, jonka mukana kulkee tieto lähetysajankohdasta.

Lähetyssäie seuraa myös lähdekoodissa 10 määritettyä kuittauskanavaa, joka välittää tiedon ACK-paketin saapumisesta. Tieto paketin saapumisesta tulee kuvan 1 pistokeabstraktiosta. ACK-paketti hylätään, jos sen järjestysnumero ei ole lähettävän ikkunan sisällä. Järjestysnumeron perusteella vastaava kehys etsitään data_queue-taulukosta ja sen tila muutetaan kuitatuksi.

```
pub struct TransmitterWindow {
    inner_window: Window,
    socket: Arc<SocketAbstraction>,
    events_sender: Sender<ConnectionEventType>,
    ack_receiver: Receiver<u32>,
    data_queue: Vec<Option<QueueFrame>>,
}
```

Lähdekoodi 10: Lähettävän ikkunan rakenne

Valikoiva toisto ARQ

Valikoiva toisto ARQ (engl. selective repeat) on tehokkaampi verrattuna toisiin ARQ-protokolliin, joista yksi esimerkki on Stop-and-wait ARQ.

Stop-and-wait ARQ:ta käyttävä lähettäjä jää odottamaan jokaisesta paketista kuittausta ennen kuin siirtyy lähettämään seuraavaa pakettia. Vastaavasti vastaanottaja hylkää kaikki paketit, jotka eivät ole seuraava paketti järjestyslukujen perusteella [6].

Valikoivassa toistossa lähettäjä lähettää kaikki ikkunan sisällä olevat paketit ja odottamaan vastaanottajan kuittauksia. Mikäli lähettäjä vastaanottaa kuittauksia ikkunan etupäästä, lähettäjä siirtää ikkunaa eteenpäin ja lähettää paketit, jotka ovat nyt ikkunan sisällä. Tämä jatkuu niin kauan, kunnes kaikki paketit on lähetetty puskurista.

Vastaanottaja vastaanottaa kaikki paketit, joiden järjestysnumero on ikkunan sisällä, vaikka ne tulisivat perille väärässä järjestyksessä. Pakettien data tallennetaan puskuriin järjestysnumeron määrittämään indeksiin. Mikäli vastaanottaja vastaanottaa paketteja, jotka ovat ikkunan etupäässä, ne siirretään ylemmille kerroksille käsiteltäväksi ja ikkunaa siirretään eteenpäin.

Tässä toteutuksessa käytettiin valikoivaa toistoa, sillä se on tehokas tiedonsiirron kannalta, mutta se vaatii datan puskuroinnin. Puskurointi vaatii huomattavaa muistin käyttöä, mitä ei saata olla pienissä laitteissa. Tämä toteutus tehtiin sillä oletuksella, että alustana on tietokone ja muistia on riittävästi.

2.4 Viestien käsittely

Protokolla käsittelee vain binääridatan lähettämistä. Monimutkaisten tietorakenteiden, kuten objektien lähettäminen protokollan kautta täytyy tehdä binäärikoodauksen kautta. Verkkosivustoissa käytetään usein tekstipohjaisia tietomuotoja, kuten HTML:ää ja JSON:ia. Nämä tietomuodot eivät sovellu reaaliaikaiseen kommunikaatioon, sillä niiden mukana tulee ylimääräisiä merkkejä, joita käytetään skeeman välitykseen. Erikoistuneet reaaliaikaiset sovellukset eivät tarvitse skeematietoja tiedon lukemiseen, sillä osapuolien välillä liikkuva tieto on todella tarkkaan määritelty.

Skeemapohjainen tieto

Skeemapohjaista dataa kannattaa hyödyntää silloin, kun vastaanottajalla ei ole tarkkaa tietoa datan sisältävästä tiedoista. Tästä hyvä esimerkki on konfiguraatiodata, joka usein koostuu kentistä ja niiden arvoista. Useat konfiguraatiomahdollisuudet eivät ole pakollisia ja sen takia niille on määritelty oletusarvot. Konfiguraatio voi sisältää tuhansia eri vaihtoehtoja ja ne on nimetty sovituilla termeillä.

Mikäli konfiguraatiotietoja haluttaisiin siirtää verkon yli, skeemapohjainen data soveltuu tähän parhaiten. Näin lähettäjä pystyy kertomaan viestissä, mitä tietoja data sisältää.

Lähdekoodissa 11 ja taulukossa 4 havainnollistetaan kaksi tapaa koodata data tiedonsiirtoa varten. JSON-data on helposti luettavaa, mutta se sisältää ylimääräisiä merkkejä, joita tietokone ei tarvitse. Binääritieto taas sisältää kaiken tarvittavan tiedon, joka riittää tietokoneelle, mutta tekee siitä vaikeasti luettavaa ihmiselle. Binääritieto vie huomattavasti vähemmän tilaa kuin tavallinen JSON-data.

Lähdekoodin 11 muoto vie 26 tavua, kun taas taulukon 4 vie 19 tavua. Tämä ero suurenee huomattavasti, kun tiedon määrä kasvaa.

```
{
    "name": "Joonas Kajava"
}
```

Lähdekoodi 11: JSON-data.

Taulukko 4: Binääridata, joka sisältää saman tiedon kuin lähdekoodissa 11.

00000100	01101110	01100001	01101101	01100101
00001101	01001010	01101111	01101111	01101110
01100001	01110011	00100000	01001011	01100001
01101010	01100001	01110110	01100001	

Tässä toteutuksessa on käytetty bincode-kirjastoa binääridatan muodostukseen.

Ennalta määritelty data

Reaaliaikaisissa sovelluksissa pitäisi aina pyrkiä siirtämään tietoa ennalta määritetyssä muodossa, jotta voidaan välttää ylimääräisen tiedon siirtämisen verkon yli.

Tässä toteutuksessa on käytetty binääridatamuotoa, jossa ensimmäinen tavu kertoo tiedon muodon. Olkoon tämä ensimmäinen tavu nimeltään jatkossa ”toimintotieto”. Taulukossa 5 esitetään esimerkki pelaajan sijainnista binäärimuodossa. Tämä binääritieto sisältää pelaajan tunnuksen ja koordinaatit minimaalisessa muodossa.

Taulukko 5: Pelaajan sijainti binääritietona.

00000001	00000010	00000011	00000100

Vastaanottaja lukee viestin toimintotiedon ja valitsee sen perusteella oikean strategian viestin käsittelyyn. Tässä toteutuksessa toimintotieto on yksi tavu, joka sallii 255 eri toimintoa. Mikäli tämä ei riitä, voidaan käyttää yhtä toimintoa skeemadatan siirtämiseen, mikä käytännössä avaa mahdollisuuden käyttää loputtoman määrän toimintoja. Toinen vaihtoehto on suurentaa toimintotietoa kahteen tai neljään tavuun.

Demossa, joka käsitellään myöhemmin luvussa 3, käytetään neljää toimintoa: String, FileInfo, ResponseToFileInfo ja FileData.

2.5 Virheenkorjaus

Saapuneiden tietojen varmistukseen käytetään tarkistussummaa. Saapuneen paketin sisällöstä muodostetaan deterministisellä algoritmilla luku, jota verrataan paketin mukana tulleeseen tarkistussummaan. Mikäli summat eivät vastaa toisiaan, voidaan olettaa, että paketti on korruptoitunut ja vastaanottaja lähettää lähettäjälle NACK-paketin. Kyseinen paketti vaatii lähettäjää lähettämään korruptoituneen paketin uudestaan. [7.]

Tarkistussumma

UDP-paketissa tarkistussumma koostuu kahdesta tavusta, joka esiintyy paketin rakenteessa juuri ennen dataa. Taulukko 6 havainnollistaa tarkistussumman (purppura) ja datan (vihreä) sijainnin paketissa.

Taulukko 6: UDP-paketti Hex-muodossa, missä tarkistussumma merkitty purppuralla ja vastaavasti data vihreällä.

18	00	00	00	60	03	82	28
00	1c	11	80	00	00	00	00
00	00	00	00	00	00	00	00
00	00	00	01	00	00	00	00
00	00	00	00	00	00	00	00
00	00	00	01	30	39	30	39
00	1c	c2	a7	00	00	00	01
02	00	0c	08	00	68	65	6c
6c	6f	20	77	6f	72	6c	64

Lähdekoodissa 12 muodostetaan 16-bittinen tarkistussumma. data-taulukon sisältämät tavut ryhmitetään 16-bittisiksi numeroiksi, minkä jälkeen ne summataan yhteen. Vaikka lopullinen tarkistussumma täytyy olla 16-bittinen, lukujen summa tallennetaan väliaikaisesti 32-bittiseen numeroon. Tämä täytyy tehdä sen takia, jotta summaoperaatiossa ei tapahdu ylivuotoa. Ilman 32-bittisen numeron apua olisi mahdollista, että summattu numero ylivuotaa, kun se ylittää luvun 2^{16} - 1. [8.]

```rust
fn calculate_checksum(data: &[u8]) -> u16 {
    let mut sum = 0u32;
    for pair in data.chunks(2) {
        let mut checksum = 0u16;
        checksum += u16::from(pair[0]) << 8;
        checksum += u16::from(pair.get(1).cloned()
        .unwrap_or_default());
        sum += checksum as u32;
    }
    while sum > 0xffff {
        let carry = sum >> 16;
        sum &= 0xffff;
        sum += carry;
    }
    !sum as u16
}
```

Lähdekoodi 12: Tarkistussumman muodostaminen

C-ohjelmointikielessä kokonaisluvun ylivuotoa ei ole määritelty. Tämä voi ja on johtanut useisiin tietoturvaongelmiin. Ylivuotoa voi olla vaikea havaita, sillä C-ohjelma jatkaa toimintaa, vaikka ylivuoto tapahtuu.

Rust-ohjelmointikielessä kokonaisluvun ylivuoto on määritelty. Ylivuoto johtaa aina ohjelman paniikkiin, joka Rust-ohjelmointikielessä viittaa virheeseen, josta ei voi palautua. Paniikki johtaa aina prosessin sammumiseen [1, luku 9.3]. Tämä käytännössä poistaa ylivuodon aiheuttamat tietoturvaongelmat [1, luku 3.2].

Lähdekoodissa 13 näytetään, miten calculate_checksum-aliohjelmassa käytetty data muodostetaan. Tarkistussumma sisältää lähettäjän ja vastaanottajan IP-osoitteet, portit, protokollan (numeraalinen arvo on 17), UDP-osuuden koon ja lopuksi kuorman. [9; 10.]

```rust
#[test]
fn checksum() {
    let ip = 1u128.to_be_bytes();
    let port = 12345u16.to_be_bytes();
    let protocol = 0x0011u32.to_be_bytes();
    let payload = [
        0x00, 0x00, 0x00, 0x01,
        0x02, 0x00, 0x0c, 0x08,
        0x00, 0x68, 0x65, 0x6c,
        0x6c, 0x6f, 0x20, 0x77,
        0x6f, 0x72, 0x6c, 0x64,
    ];
    let udp_packet_length = 28u16;
    let udp_packet_length = udp_packet_length.to_be_bytes();
    let mut data = vec![];
    data.extend_from_slice(&ip);
    data.extend_from_slice(&ip);
    data.extend_from_slice(&protocol);
    data.extend_from_slice(&udp_packet_length);
    data.extend_from_slice(&port);
    data.extend_from_slice(&port);
    data.extend_from_slice(&udp_packet_length);
    data.extend_from_slice(&payload);
    let checksum = calculate_checksum(&data);
    assert_eq!(checksum, 0xc2a7);
}
```

Lähdekoodi 13: Tietojen kasaaminen tarkistussummaa varten.

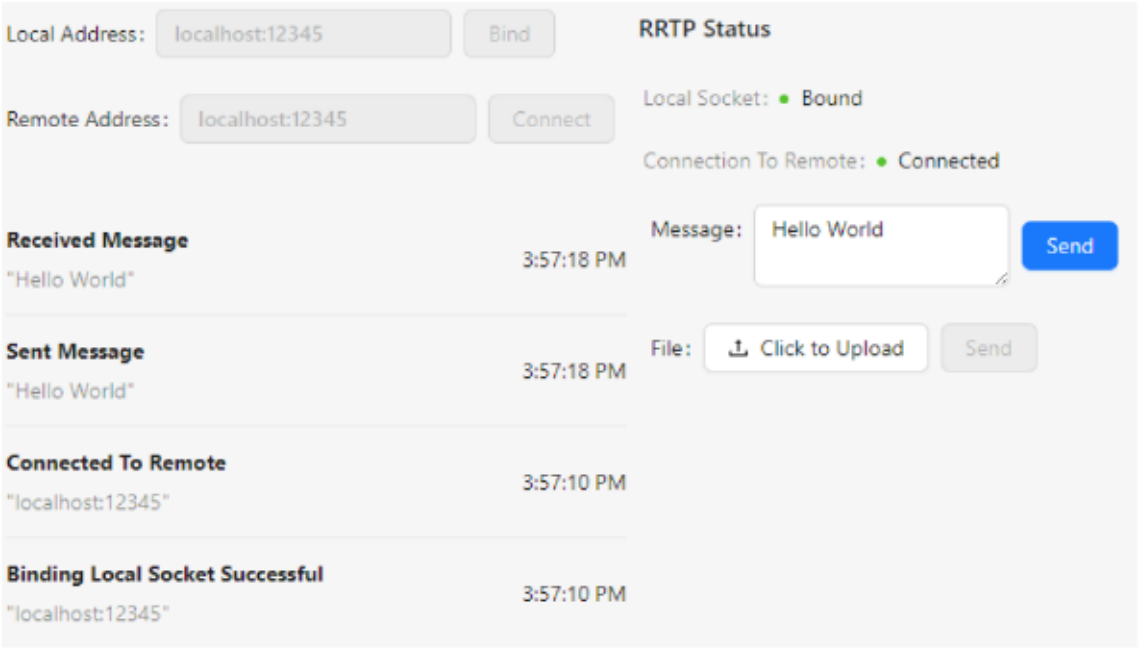
Modernien tietokoneiden verkkokortti hoitaa tämän tarkistussumman käsittelyn [11].

3 Demosovellus

Osana tätä työtä toteutettiin esimerkkiohjelma, jolla pystyy luotettavasti siirtämään tiedostoja ja viestejä verkon yli. Demosovellus on rakennettu käyttäen Tauri- ja React-kirjastoja. Koska Rust-ohjelmointikieli on vielä nuori ja kattavia käyttöliitty-

mä kirjastoja ei ole montaa [12]. Egui- ja Iced-projektit olivat hyviä vaihtoehtoja Taurille, mutta niiden dokumentaatio ja yleinen kypsyys eivät olleet riittäviä tämän projektin tarpeisiin.

Demosovellus on rakennettu käyttäen Tauri-ohjelmistokehystä, joka on Electronin kaltainen ohjelmistokehys. Tauri eroaa Electronista siinä, että Electron suoritetaan Node.js runtime -ympäristön päällä ja tarjoaa version Chromium-selaimesta käyttöliittymän piirtämiseen. Tauri ei tarjoa selainta vaan Rust-prosessi luo ikkunan käyttäen TAO-ohjelmistokirjastoa, minkä jälkeen WRY-ohjelmistokirjasto luo WebView-elementin, joka kutsuu käyttöjärjestelmän tarjoamaa selainta näyttämään kuvan 3 mukaisen käyttöliittymän. [13.]



Kuva 3: Demosovelluksen käyttöliittymä.

Demosovelluksesta löytyy kaksi kenttää osoitteiden asettamiseen. Lista joka pitää kirjaa tapahtumista. Tilannepaneeli joka näyttää yhteyden tilan: Kaksi kenttää viestien ja tiedostojen lähettämiseen.

3.1 Riippuvuudet

Demosovelluksessa on mahdollista käyttää crates-palvelun Rust-kirjastoja ja npm-palvelun javascript-paketteja. Paketit mahdollistavat sovelluksen nopean kehityksen ja tarjoavat vakaita ratkaisuja eri ongelmiin. Usein pakettien käyttäminen on

erittäin suositeltua, sillä niiden dokumentaatio ja valmiit ratkaisut ovat hyödyksi kehityksessä ja ylläpitämisessä.

Taulukko 7: Rust-riippuvuudet

Nimi	**Käyttötarkoitus**
tauri	Käyttöliittymän piirtäminen
serde/bincode	Tiedon serialisointi ja deserialisointi
fern/log/humantime	Lokitiedostojen kirjoittaminen
typeshare	Typescript-rajapintojen luonti Rust-tyypeistä
infer	Tiedostotietojen lukeminen
thiserror/anyhow	Virheilmoituksien luonti ja käsittely

Taulukko 8: Npm-riippuvuudet.

Nimi	**Käyttötarkoitus**
ahooks	Hyödyllisiä react-koukkuja
antd	Käyttöliittymäelementit
dayjs	Aikatietojen käsittely
pretty-bytes	Tiedostokoon muuntaminen ihmisluettavaksi
rc-virtual-list	Virtuaaliset listat
react/react-dom	Käyttöliittymän hallinta ja piirtäminen
recoil	Käyttöliittymän tilan hallinta
typescript	Tyyppitietojen lisääminen javascriptiin
vite	Front-end ohjelmiston prosessointi

3.2 Sovelluksen käynnistysvaihe

Demosovellus alkaa main.rs-tiedoston main-aliohjelmasta. Se suorittaa 5 tärkeää toimintoa, jotka ovat lokituskonfiguraatio, sovellustilan alustaminen, lokiseurantasäikeen käynnistys, IPC-rajapintojen määritys ja lopuksi käyttöliittymän käynnistys.

Lokituskonfiguraatio tapahtuu setup_logger-aliohjelman kautta, mikä määrittelee lähdekoodin 15 mukaiset asetukset. Lähdekoodissa 14 on havainnollistettu esimerkki lokitapahtumasta, josta ilmenee selkeästi tapahtuma-aika, viestin taso, tapahtuman moduuli ja itse viestin sisältö.

```
[2024-03-29T13:57:18Z INFO messanger::connection_processor]
Processing connection event:
ReceivedCompleteMessage([0, 72, 101, 108, 108, 111,
32, 87, 111, 114, 108, 100])

```

Lähdekoodi 14: Esimerkki lokitapahtumasta.

Lähdekoodi 14 esittää mahdollisen lokitapahtuman. Sovellus voi kirjoittaa lokitapahtuman käyttämällä log-kirjaston tarjoamia makroja, kuten lähdekoodin 17 rivillä 12 on tehty.

```
fern::Dispatch::new()
.format(|out, message, record| {
    out.finish(format_args!(
        "[{} {} {}] {}",
        humantime::format_rfc3339_seconds(SystemTime::now()),
        record.level(),
        record.target(),
        message
    ))
})
.level(log::LevelFilter::Debug)
.chain(std::io::stdout())
.apply()?;

```

Lähdekoodi 15: Lokituskonfiguraatio

Sovellustila alustetaan Builder-rakenteen setup-aliohjelmalla, jonka tunniste on kuvattu lähdekoodissa 16. Aliohjelman tunniste sisältää tiedot sen vaatimuksista, joita täytyy noudattaa sen sisällä. Näistä vaatimuksista Send on erityisen tärkeä, sillä se takaa muistiturvallisuuden säikeiden välillä. Tämä mahdollistaa kanavien käytön sulkeumassa ilman muistiongelmia [1, luku 8.2].

```
pub fn setup<F>(mut self, setup: F) -> Self
where
F: FnOnce(&mut App<R>) ->
Result<(), Box<dyn std::error::Error>> + Send + 'static

```

Lähdekoodi 16: Setup-aliohjelman tunniste.

setup-sulkeumassa, joka on kuvattu lähdekoodissa 17, luodaan lokituskanava riveillä 1–4. Nämä kanavat mahdollistavat viestimisen säikeiden välillä ilman lukkoja. Ilman kanavia säikeiden välillä jaetut muuttujat täytyisi lukita käytön ajaksi lukolla, joka estää muita säikeitä käyttämästä kyseistä muuttujaa. Tätä kanavaa pitkin Rust-prosessi pystyy lähettämään käyttöliittymään viestejä mistä tahansa säikeestä. Kanavan viestejä pystyy vastaanottamaan vain yhdessä paikassa, mutta viestien lähetyksessä ei ole tätä rajoitusta [1, luku 16.2]. Rivillä 7 luodaan säie viestien lukemista varten, mikä odottaa viestejä kanavasta ja saatuaan viestin välittää sen käyttöliittymään IPC-viestillä. Rivillä 5 luodaan sovelluksen tilarakenne, joka vastaanottaa lokikanavan lähettävän pään. Tämän jälkeen kutsutaan Tauri-sovelluskehystä hallitsemaan tätä tilaa.

```
let (log_sender, log_receiver): (
    Sender<LogSuccessMessage>,
    Receiver<LogSuccessMessage>,
) = std::sync::mpsc::channel();
let app_state = AppState::new(log_sender);
let handle = app.handle();
tauri::async_runtime::spawn(async move {
    loop {
        let message = log_receiver.recv().unwrap();
        match handle.emit_all("log", message) {
            Ok(_) => {}
            Err(e) => error!("Failed to emit log message: {}", e),
        }
    }
});
app.manage(app_state);
```

Lähdekoodi 17: setup-sulkeuma.

Sovelluksen tila sisältää yhteyden tilan, viestien tilan ja viitteen lokikanavan lähettävään päähän. Lähdekoodin 18 riveillä 11 ja 13 on huomattava, että nämä ovat Mutex<...> primitiivin sisällä. ConnectorState ja MessageState eivät toteuta Send-ominaisuutta, joka löytyy setup-aliohjelman tunnisteesta. Mutex<...> primitiivi toimii lukkona, jonka avulla Send-vaatimus täyttyy. Sender<...> toteuttaa Send-ominaisuuden, joten sitä ei tarvitse erikseen laittaa Mutexin sisälle.

```
1  struct ConnectorState {
2      pub connector: Option<ConnectionManager>,
3      message_sender: Option<SyncSender<Vec<u8>>>,
4  }
5  #[derive(Default)]
6  struct MessageState {
7      pub outgoing_file: Option<FileInfo>,
8      pub incoming_file: Option<FileInfo>,
9  }
10 struct AppState {
11     pub connector_state: Mutex<ConnectorState>,
12     pub log_sender: Sender<LogSuccessMessage>,
13     pub message_state: Arc<Mutex<MessageState>>,
14 }
```

Lähdekoodi 18: Sovellustilan rakenne.

3.3 Prosessien välinen kommunikaatio

Tauri-sovelluskehys hyödyntää IPC-viestejä käyttöliittymän ja Rust-prosessin väliseen kommunikointiin.

Tauri-sovelluskehys sisältää kaksi IPC-primitiiviä: tapahtumat ja komennot. Tapahtuma on yksisuuntainen viesti, jota käytetään pääosin ohjelmatilan muutoksien kommunikointiin. Lähdekoodissa 17 rivillä 10 lähetetään tapahtuma, jonka käyttöliittymä vastaanottaa. Komento on Web API:n kaltainen viestintämenetelmä, jossa tieto siirtyy JSON-RPC-kaltaisella protokollalla. Tämä mahdollistaa monimutkaisien tietorakenteiden käytön viestinnässä, sillä ne serialisoidaan JSON-muotoon [13].

3.4 IPC-komennot

IPC-komennot merkitään #[tauri::command]-määritteellä. Aliohjelma täytyy olla merkitty kyseillä määritteellä, jotta sitä pystyy käyttämään lähdekoodin 19 määrityksessä. Tauri pystyy tarjoamaan viitteet muun muassa sovellustilaan ja sovelluskahvaan käyttämällä riippuvuusinjektio (engl. Dependency Injection) suunnittelutapaa. Tämä suunnittelutapa on todella yleinen useissa sovelluskehyksissä, kuten ASP.NET Core -kehitysalustassa [14]. Tämä suunnittelutapa ei kuitenkaan ole yleinen Rust-kielessä, sillä se vaatii aliohjelmien suorittamista dynaamisilla kutsuilla (engl. dynamic method invocation). ASP.NET Coressa Reflection-toiminnallisuus tarjoaa dynaamisen kutsumisen, mikä tekee tästä suunnitteluta-vasta käytännöllisen. Tauri-sovelluskehys toteuttaa tämän käyttämällä Procedu-ral Macros -toimintoa.

Lähdekoodi 19 määrittelee 5 IPC-komentoa, joita käytetään käyttöliittymän ja Rust-prosessin välisessä kommunikaatiossa. Näitä komentoja voidaan myöhemmin kutsua käyttöliittymästä käsin käyttäen invoke-aliohjelmaa, kuten lähdekoodissa 20 on esitetty.

```
.invoke_handler(tauri::generate_handler![
    bind,
    connect,
    send_message,
    send_file_info,
    respond_to_file_info
])
```

Lähdekoodi 19: IPC-komentojen määritys.

```
invoke<LogSuccessMessage>("bind", {address: localAddress})
.then((result) => {
    setLog(result);
    setConnectionStatus((prev) => ({...prev, local: true}));
})
.catch((err: LogErrorMessage) => {
    setLog(err);
    setConnectionStatus((prev) => ({...prev, local: false}));
});
```

Lähdekoodi 20: bind-komennon kutsuminen.

Bind-komento liittää verkkosovittimen address-parametrin osoitteeseen. Liittämisen hoitaa ConnectionManager-rakenne, joka käynnistää myös tarvittavat säikeet viestien ja kuittauksien vastaanottoa varten sekä säikeen viestien lähettämisetä varten. Lopuksi bind-komento käynnistää säikeen, joka kuuntelee ConnectionManagerin tuottamia tapahtumia. bind-komennossa käynnistetty säie ohjaa ConnectionManagerin tuottamat tapahtumat ConnectionProcessor-rakenteeseen, joka lajittelee tapahtumat niiden tyypin mukaan ja suorittaa tarvittavan strategian tapahtuman käsittelyyn.

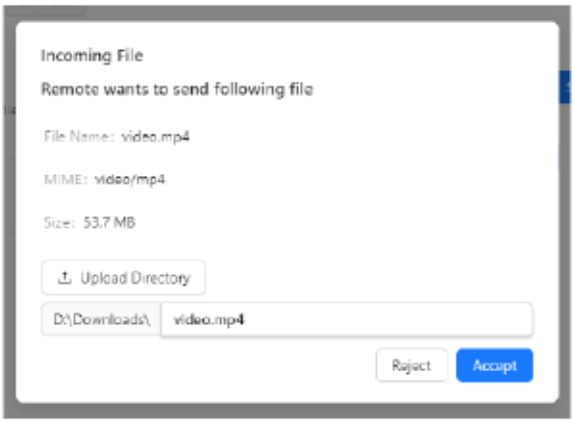
Connect-komento vastaanottaa osoitteen ja sovellustilan, joka tulee Taurin DI-järjestelmästä. Komennon tarkoitus on valmistella sovellustila viestien lähettämistä varten. Lähdekoodissa 21 rivillä 4 yhteyden tila lukitaan, joka on Mutex-rakenteen toiminto. lock-aliohjelma lukitsee connector_state-muuttujan ja palauttaa MutexGuard<ConnectorState>-rakenteen. Tämä suoja sallii muutoksien tekemisen ConnectorState-rakenteeseen, vaikka viite ei normaalisti salli muutoksia. Tätä piirrettä kutsutaan nimeltä sisäinen muuttuvuus (engl. *Interior Mutability*). Yhteystila on lukittuna niin kauan kun MutexGuard<ConnectorState> on käytössä.

```
1  #[tauri::command]
2  pub fn connect(address: &str, state: State<AppState>)
3  -> LogMessageResult {
4  let app_state_lock = state.connector_state.lock().unwrap();
5  let connector = &app_state_lock.connector;
6  let result = connector
7  .as_ref()
8  .ok_or(LogErrorMessage::LocalSocketNotBound)?
9  .connect(address);
10 result
11 .map(|_| LogSuccessMessage::ConnectedToRemote(address.to_string()))
12 .map_err(|e| LogErrorMessage::ConnectionError(e.to_string()))
13 }
```

Lähdekoodi 21: connect-komento.

send_message, send_file_info ja respond_to_file_info ovat komentoja, jotka lähettävät tietoja verkon yli. send_message lähettää yksinkertaisen viestin vastaanottajalle. send_file_info lähettää vastaanottajalle tiedoston metatiedot: tiedoston nimi, tiedoston MIME-tyyppi ja koko tavuina. Näiden tietojen avulla vastaanotta-

ja pystyy valmistautumaan tulevaan tiedostoon. Kuva 4 havainnollistaa, miltä tuleva tiedosto näyttää vastaanottajalle. Vastaanottaja pystyy kuvan 4 mukaisella ikkunalla valitsemaan, mihin kansioon kyseinen tiedosto tullaan tallentamaan tai vaihtoehtoisesti hylkäämään tulevan tiedoston. Valinnan tekeminen laukaisee respond_to_file_info-IPC-komennon, joka ilmoittaa lähettäjälle valinnasta. Mikäli valinta on myönteinen, lähettäjä alkaa välittömästi lähettämään tiedostoa verkon yli. Jos valinta on kielteinen, lähettäjä saa ilmoituksen hylätystä tiedostosta.



Kuva 4: Tulevan tiedoston tiedot.

4 Suorituskyky

Kehityksen aikana huomioitiin myös toteutuksen suorituskyky. Rust-kieli on itsestään jo todella suorituskykyinen, jopa C-kieleen verrattuna. Nollakustannusabstraktiomalli sallii ohjelmoinnin käyttämällä korkeita abstraktioita ilman suorituskyky haittaa [15]. Huomattava osa suorituskykyongelmista ei johdu kielestä, vaan käytetyistä algoritmeista ja tietorakenteista. Tämän takia tehtiin suorituskykyanalyysejä kaikista kriittisimpiin tietorakenteisiin.

4.1 Tietorakenteet

RRTP-kirjasto käsittelee huomattavan määrän dataa tietorakenteiden avulla, joten tehokkaiden tietorakenteiden käyttö on tärkeää. Kirjaston täytyy pystyä käsittelemään tehokkaasti listoja, jotka sisältävät tuhansia alkioita.

Mittauksessa mitattiin tavallisen taulukon, kasavaratun taulukon, Vec-, VecDeque- ja LinkedList-rakenteiden suoriutumista, kun tietorakenteen alkioita siirrettiin vasemmalle. Jokainen tietorakenne täytettiin 100 000 numerolla ja alkioita siirrettiin

vasemmalle parhaiten siihen sopivalla tavalla. Mittaus tehtiin käyttäen criterion-kirjastoa ja 100:n näytemäärällä.

Taulukon 9 tulosten perusteella kasataulukko oli paras vaihtoehto mitatuista tietorakenteista. Tästä huolimatta projektissa käytettiin Vec-tietorakennetta, sillä tarvittavan taulukon koko ei ole tiedossa kääntämisen aikana. Vec-tietorakenne sallii sen koon muuttumisen ohjelman suorituksen aikana.

Taulukko 9: Tietorakenteiden nopeus alkioiden siirrossa vasemmalle.

Tietorakenne	**Tulos**
Kasataulukko	61,603 µs
Taulukko	72,929 µs
Vec	124,49 µs
VecDeque	176,03 µs
LinkedList	4,5944 ms

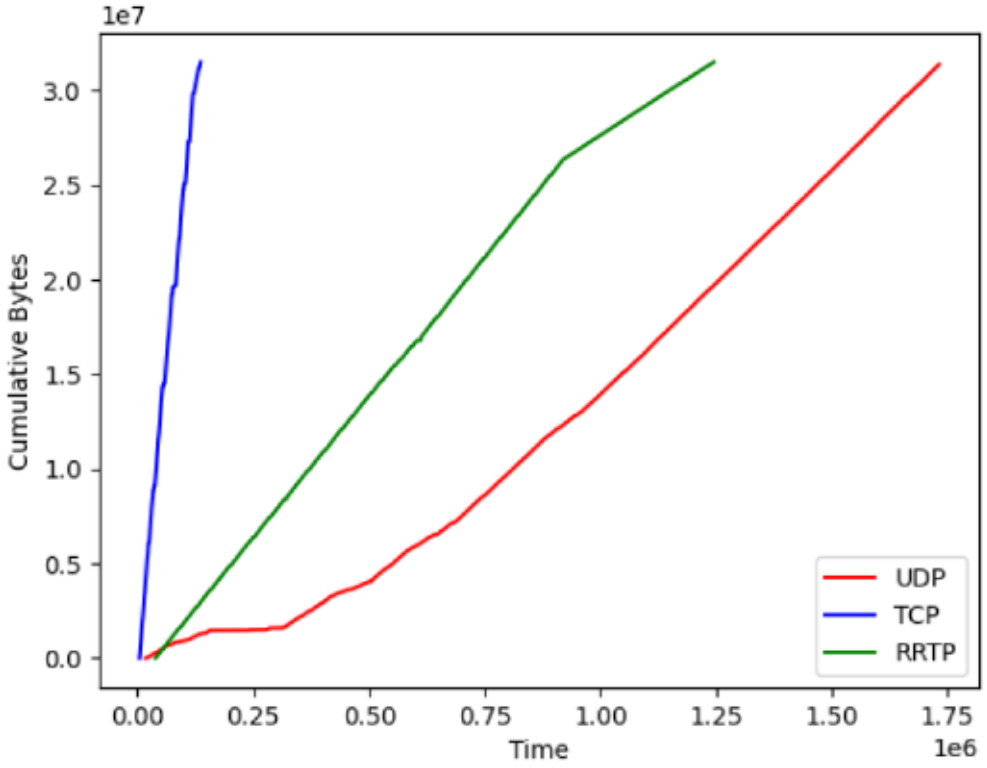
Tuloksista voidaan myös todeta, että LinkedList-tietorakennetta tulisi käyttää vain harvoissa tilanteissa. Rust-kielen standardikirjaston dokumentaatio vahvistaa tämän väitteen [3, *LinkedList*]. Kääntäjä pystyy tekemään erinomaisia optimointeja muille tietorakenteille ja tietokoneen prosessori pystyy hyödyntämään välimuistia tehokkaammin.

4.2 Tiedostojen siirtäminen

Tiedonsiirron testaamisessa käytettiin 30 MB:n kokoista videotiedostoa, jotta tiedoston eheys olisi mahdollisimman helppo tarkistaa. Tässä testissä verrattiin RRTP-kirjaston, UDP:n ja TCP:n tiedonsiirtokykyä käyttäen silmukkaosoitetta.

Kuvasta 5 voidaan päätellä, että RRTP-kirjaston suorituskyky on kohtalaisen hyvä verrattuna UDP:hen ja TCP:hen. TCP on näistä selvästi nopein, mikä johtunee siitä, että TCP:ssä puskurin käsittely on lähes automaattinen ja puskuri kasvaa tarvittaessa. UDP:n ja RRTP-kirjaston kanssa käytettiin 128 tavun kokoista puskuria, mikä selittää näiden kahden tuloksen hitauden. Tulosten mukaan UDP on hidas lähetyksen alussa, mutta nopeutuu ajan kuluessa. Työn aikarajoitusten takia jatkotutkimusta tiedosiirron hitaudesta ei toteutettu.

Kokonaisuudessaan RRTP-kirjasto suoriutuu suhteellisen hyvin tehtävästään, mut-



Kuva 5: Tiedonsiirtovertailun tulokset.

ta huomiota täytyy kiinnittää sen tehokkuuteen isoja tiedostoja siirrettäessä. Kuvasta 5 nähdään, että kirjaston suorituskyky pienenee, mitä enemmän tietoa on siirretty. Tämä johtunee siitä, että kirjaston sisäinen puskuri täytyy ja alkioiden käsittely vie enemmän aikaa.

Silmukkaosoitteen takia tämä testi ei kuitenkaan vastaa täysin todellisuutta. Testin tarkoitus oli pääasiassa seurata tietorakenteiden käsittelyn tehokkuutta ilman verkon aiheuttamaa häiriötä.

5 Työkalut ja prosessit

Tässä työssä käytettiin alalla laajalti hyväksyttyjä sovelluskehitystapoja. Osa työkaluista ja menetelmistä on suhteellisen kokeellisia, mutta käytettiin kuitenkin vain kehittäjäkokemuksen parantamiseen, eikä niitä käytetty työn kannalta kriittisissä osissa.

5.1 Versionhallinta

Tässä työssä käytettiin git-versionhallintaa, joka oli github-palvelun isännöimä. Git on toimialalla erittäin yleinen ja tärkeä työkalu lähdekoodin jakamiseen ja versionhallintaan.

Vetopyyntöjen(engl. Pull Request) tekeminen on yleinen ja suositeltu tapa liittää muutokset pääharaan. Tämä johtuu siitä, että kun projektissa on useita ihmisiä tekemässä, jokainen muutos vaatii toisten kehittäjien hyväksynnän. Kun kriteerit hyväksymiseen on toteutunut, muutos liitetään päähaaraan.

Työn toteutuksen aikana projektiin tuli noin 5 suurta refaktorointia. Näitä varten käytettiin useita haaroja, joiden avulla pystyttiin pitämään päähaara aina toimivassa tilassa.

Projektin lähdekoodi on saatavilla github.com-palvelusta osoitteesta: https://github.com/JoonasKajava/ReliableRealTimeTransportProtocol.

5.2 Kehitysympäristö

Alustana toimi taulukon 10 mukainen pöytätietokone. Editorina oli pääsääntöisesti käytössä ohjelmistokehitysympäristö RustRover, jossa oli muutama tärkeä lisäosa, kuten IdeaVim, Grazie Pro, SonarLint ja WakaTime. UDP-pakettien seurantaan käytettiin WireShark-ohjelmistoa.

Taulukko 10: Kehitysalustan tiedot.

Ohjelmisto	**Versio**
Käyttöjärjestelmä	Microsoft Windows 10.0.19045 Pro
Rust-kääntäjä	rustc 1.75.0 (82e1608df 2023-12-21)
Node.js	v18.8.0

6 Yhteenveto

Tavoitteena tässä opinnäytetyössä oli tutustua Rust-ohjelmointikieleen ja reaaliaikaisten kommunikaatiojärjestelmien luontiin. Työn tuloksena syntyi RRTP-kirjasto, joka hyödyntää liukuvaa ikkunaa ja muita toimintoja luotettavan yhteyden muodostamiseen.

RRTP-kirjasto pystyy siirtämään viestejä ja tiedostoja ilman tiedon korruptiota. Lähdekoodi on suurimmaksi osaksi idiomaattista Rust-koodia, ja käyttöliittymän React-koodi käyttää ajankohtaisia suosituksia ja menetelmiä. Koodi on jaettu yksinkertaisiin moduuleihin ja suunniteltu niin, että laajennukset tai muutokset ovat helppoja toteuttaa.

Jatkokehityspolku on selkeä. Protokollaan voidaan toteuttaa lisää TCP:n hyödyntämiä tekniikoita, kuten dynaaminen aikakatkaisu paketeille ja dynaaminen koko liukuvalle ikkunalle. Nämä kaksi tekniikkaa saadaan helposti lisättyä, koska mahdolliset muutokset otettiin huomioon arkkitehtuurin suunnittelussa.

Lopputuloksena syntyi vakaa ja toimiva kirjasto reaaliaikaiselle kommunikaatiolle. Kirjasto kirjoitettiin kokonaan käyttäen turvallista Rust-kieltä, joka takaa muistiturvallisuuden samalla tavalla kuin ohjelmointikielet, jotka käyttävät automaattista roskienkeräystä. Lisäksi Rust-kielen vaativa syntaksi ja tyyppimallit takaavat, että ohjelmassa ei voi olla epäkelpoisia tiloja. Tätä filosofiaa pystyttiin myös jatkamaan käyttöliittymän puolella käyttämällä tarkkoja TypeScript-sääntöjä ja TypeShare-kirjastoa, jotta TypeScript-tyypit saatiin vastaamaan täsmälleen Rust-tyyppejä.

Protokollan kehityksessä tutustuttiin säikeiden turvalliseen käyttöön ja miten Rust-kieli auttaa tässä. Tärkeänä osana oli myös toteuttaa toimiva käyttöliittymä, joka pystyy hyödyntämään Rust-kielen vahvuuksia. Tämä on erityisen tärkeää, sillä uskon vahvasti, että Rust-kieli on tulevaisuuden kieli. Sitä käytetään jo muun muassa Linuxin ja Windowsin kernelissä. Rust-kielen turvallisuus, tehokkuus ja modernit ominaisuudet tekevät siitä lähes täydellisen kielen, jolla on mahdollisuus korvata C ja C++ tulevaisuudessa.

Viitteet

[1] Klabnik, S., Nichol, C. 2022. The Rust Programming Language. Verkkoaineisto. <https://doc.rust-lang.org/book>. Luettu 28/03/2024.

[2] Kumar, S., Rai, S. 2012. Survey on Transport Layer Protocols: TCP & UDP. International Journal of Computer Applications.

[3] The Rust Standard Library. Verkkoaineisto. <https://doc.rust-lang.org/std/index.html>. Luettu 13/05/2024.

[4] Adiga, H. 2007. How to write endian-independent code in C. Verkkoaineisto. <https://developer.ibm.com/articles/au-endianc/>. Luettu 21/02/2024.

[5] Ivicevic, C. 2022. 28 Days of Rust. Verkkoaineisto. <https://medium.com/comsystoreply/28-days-of-rust-part-2-composition-over-inheritance-cab1b106534a>. Luettu 25/02/2024.

[6] Gaurav, S. 2023. Stop and Wait ARQ. Verkkoaineisto. <https://www.scaler.com/topics/computer-network/stop-and-wait-arq/>. Luettu 27/02/2024.

[7] Khan Academy. User Datagram Protocol (UDP). Verkkoaineisto. <https://www.khanacademy.org/computing/computers-and-internet/xcae6f4a7ff015e7d:the-internet/xcae6f4a7ff015e7d:transporting-packets/a/user-datagram-protocol-udp>. Luettu 27/03/2024.

[8] Google. Android-lähdekoodi. Verkkoaineisto. <https://android.googlesource.com/platform/system/core/+/master/libnetutils/packet.c>. Luettu 28/03/2024.

[9] Postel, J. 1980. RFC 768 - User Datagram Protocol. Verkkoaineisto. <https://datatracker.ietf.org/doc/html/rfc768>. Luettu 28/03/2024.

[10] Protocol Numbers. Verkkoaineisto. <https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml>. Luettu 29/03/2024.

[11] Viviano, A. TCP/IP Offload Overview. Verkkoaineisto. <https://learn.microsoft.com/en-us/windows-hardware/drivers/network/tcp-ip-offload>. Luettu 27/03/2024.

[12] Are we GUI yet? Verkkoaineisto. <https://areweguiyet.com/>. Luettu 25/02/2024.

[13] Tauri App. Verkkoaineisto. <https://tauri.app/>. Luettu 29/03/2024.

[14] Larkin, K., Smith, S., Dahler, B. Dependency injection in ASP.NET Core. Verkkoaineisto. <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-8.0>. Luettu 30/03/2024.

[15] Codex, A. 2023. Understanding Rust's Zero-Cost Abstractions. Verkkoaineisto. <https://reintech.io/blog/understanding-rust-zero-cost-abstractions>. Luettu 15/05/2024.